Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5753--17-9730

Electronic Warfare *M*-on-*N* Digital Simulation Logging Requirements and HDF5: A Preliminary Analysis

Donald E. Jarvis

Advanced Techniques Branch
Tactical Electronic Warfare Division

April 12, 2017

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
12-04-2017	NRL Memorandum Report	January 2017 – February 2017

4. TITLE AND SUBTITLE

Electronic Warfare *M*-on-*N* Digital Simulation Logging Requirements and HDF5: A Preliminary Analysis

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Donald E. Jarvis

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5753--17-9730

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

10. SPONSOR / MONITOR'S ACRONYM(S)

NRL

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report documents the investigations made in a rapid evaluation of the role of HDF5 technology in logging the output of electronic warfare (EW) computer simulations. The goal was either early identification of any clear disqualifications of HDF5 or an outline for how it may be moved forward in an EW logging solution. No up-front disqualifications were identified. The distinction between representation and query notation is emphasized. Novel contributions include an abstract data model of the EW simulation logging stream, and an exploratory conformation to both the relational and HDF5 data models.

15. SUBJECT TERMS

Electronic warfare
Modeling and simulation
Data model
Database
Relational algebra
HDF5

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					Donald E. Jarvis
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	SAR	18	**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified Unlimited	Unclassified Unlimited	Unclassified Unlimited	Unlimited		202-404-7682

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

CONTENTS

ELECTRONIC WARFARE *M*-ON-*N* DIGITAL SIMULATION LOGGING REQUIREMENTS AND HDF5: A PRELIMINARY ANALYSIS

1. INTRODUCTION

HDF5 technology [Folk] has been proposed as a standard for file storage of large arrays of scientific data as well as an application programming interface (API) for accessing that data. This report documents an evaluation charrette of the potential role of HDF5 in logging the output of computer simulations of electronic warfare (EW) engagements. The goal was to either rapidly identify any *prima facie* disqualifications of HDF5 or to outline how it may be moved forward in an EW logging solution. A focus of this investigation was an analysis at the level of abstract data modeling.

2. PROBLEM STATEMENT

Here we sketch the logging requirements of an EW simulation. This account is informal and does not claim to be exhaustive. Questions are noted as they arise.

The EW simulation logging problem, stated simply, is as follows: 1) We have a collection of simulated things (entities or subsystems), where each is producing multiple streams of time series data. 2) The data are not necessarily synchronous nor of the same type. 3) We want to log these data to a file for later analysis.

To better understand the structure of the problem apart from details of implementation, we consider what an appropriate abstract data model for the problem would be. Should this data model reflect the simulation that generates it, the model implied by the query notation, or the data's inherent structure? The first two options pertain mostly to convenience. We will emphasize the data's inherent structure, confident that this can be rendered into convenient representations as needed.

The EW simulation has a collection of entities or subsystems that can log data, but we will de-emphasize these and focus instead on the data themselves as primary. It does follow that most logged data will be associated with some "owner" parent or subsystem, and this association should be recorded. Not all data will have an associated entity, e.g. parameters pertaining to the simulation run as a whole. The collection of logging entities and/or subsystems may be flat or hierarchical; what impact if any does this have on logging? Is it a substantial difference, or a mere implementation detail of the API?

A logging entity or subsystem can log asynchronously, therefore, most logged data is timestamped. A timestamp is not appropriate for all data, because some data pertains to the entity, subsystem or simulation as a whole, therefore a timestamp cannot be mandatory. Is it allowable for the same item to have both non-timestamped and timestamped values? Should the timestamp itself be considered data or an attribute? Is time only meaningful in its role of describing data associated with it, e.g. so the fact that the timestamp sequence was 0.0, 0.1, 0.2… alone (apart from its associated data) is regarded as completely uninformative?

Manuscript approved March 16, 2017.

Logged data will necessarily be of some type. An entity or subsystem can log data selected from a menu of types, but what is the extent of that menu? At one extreme it may be limited to a handful of well-known types (e.g., floats, ints, arrays of the same, strings). At the other it may allow arbitrary user-defined objects. The difference is in the availability of methods to work with the data. In the former case they are typically available in the implementation language – most programming languages have constructs for working with numbers and strings built-in for convenience. In the latter, the object must be serializable and deserializable for storage in and retrieval from the database, and the user is responsible for providing the software to work with and interpret the contents.

2.1 Hierarchical Quality

The role of hierarchy is a persistent theme. When several data all contribute to a description of the same thing, it is natural to group this data together under a common heading. Is the data inherently hierarchical? If so, in what sense? Is this relevant to logging? Is a relational database solution disqualified because it is not hierarchical?

We will make some preliminary distinctions here. Hierarchy in the data appears to be of two distinct kinds. The first involves an individual data stream's association with (or containment in) a subsystem, and containment of subsystems in an entity or larger subsystem. Hierarchy in this sense pertains to the simulation *model*. The second sense involves composite data types such as arrays, where e.g. a 2-D array is conceptualized as a container for rows, and a row is in effect a 1-D array, a container for elements. Hierarchy in this sense pertains to *nested containers* and is at work at a level somewhat independent of the simulation model.

A table in which some column values tag their row such that they can be "factored out" recursively to a tree structure can be considered a hierarchical interpretation of that table; see Figure 1 (also see the Group operator, [Date2005 p99]; and contrast the Ungroup operator, *ibid*., with the traditional transformation to first normal form, e.g. as described in [Welling p32]). On the other hand, while we can factor a table out into a tree structure, the resulting tree is not strongly hierarchical in the sense that it is still equivalent to a table; the tree is not general, but is restricted to a special structure that results from its underlying table-ness. A distinctly hierarchical tree would allow for different depths in different parts. A label-value characteristic associated with this would be that the value at more prior nodes of the tree determines the labels at later nodes. A tree derived from a table does not have this property; in a tree derived from a table, node label is determined by depth only.

The considerations above raise the question of whether they were exhaustively addressed in the development of the hierarchical database designs of the 1960's. A representative account on these legacy implementations in a recent reference [Silberschatz, Appendix E] suggests that they were not. For example, it appears to follow from the tree-structure diagram schema that all nodes at the same depth have the same type, resulting in a database with a layered structure (a structure reminiscent of the table-derived tree described above). This hierarchy is qualitatively different from the one we are interested in here. It is claimed in [Date2005 p170] that the hierarchical data model was invented after the fact of implementation; it would follow that any contemporary theory would be heavily laden with

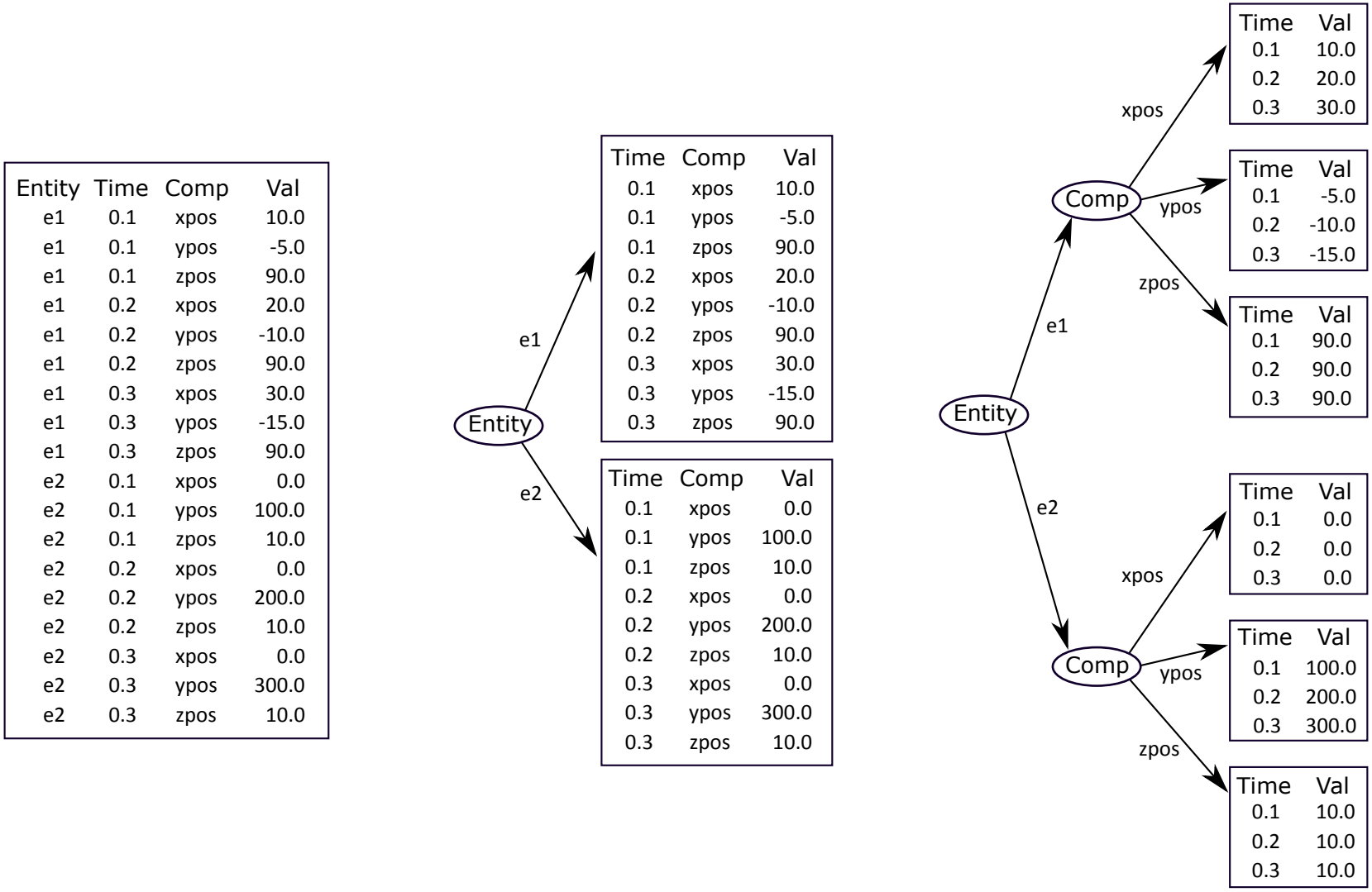

Figure 1. Two levels of "factoring" columns out of a table. From left to right are the original table, the Entity column factored out, the Entity and Comp columns factored out. The data remaining at the leaves of the tree can be readily represented in array form.

implementation details of the time. Therefore, a reconsideration of hierarchical quality as carried out in this report is appropriate.

2.2 Scoping

If the logger can be conceptualized in the context of the EW simulation as a block in a block diagram, this suggests certain advantages and limitations. Such a block will, by design, not have direct access to the internal structure of the entities and subsystems it is logging, and so any notion of hierarchical tags for the logged data will be emplaced manually. Similarly, the logging block will have convenient access to the time-series data fed to it but can only be privy to entity, subsystem, and simulation-wide initial parameters (non-timestamped data) by some other mechanism.

3. DATA MODEL OF THE LOGGING STREAM

The goal of this report is to investigate logging of EW simulations not at the level of implementation in a database management system (DBMS) or other software, but at a higher level of abstraction. This requires an abstract model of the EW logging problem. In this section we propose an abstract data model for logging EW simulations and consider numerous issues related to it.

3.1 The Logged Item

From an abstract data model viewpoint, a log is a sequence of logged *items*. We call this sequence a *logging stream*. We propose the following formal structure for an item (the definitions here are modeled in part after similar concepts in [Date2005]).

Informally, an item is a collection of label-value pairs. The values are typed, where the type is associated with the label. The labels in an item are unique.

Proceeding more formally, we begin by pairing a label and its associated type. This pair is called a *column*. A column can be paired with a value (of an appropriate type). A set of such column-value pairs is an *item*.

$$\overbrace{\left\{\left(\overbrace{\left(\underbrace{A_i}_{\text{label}}, \underbrace{T_i}_{\text{type}}\right)}^{\text{column}}, \overbrace{V_i}^{\text{value}}\right)\right\}}^{\text{set-of}}$$

A similar structure consisting of a set of columns only (without values) is called a *heading*.

An *element* in reference to items and headings is used in the set-theoretic sense as a synonym for "member," so the elements of an item are column-value pairs, and the elements of a heading are columns.

As sets, the elements of items and headings are unordered. We emphasize that in the context of headings and items, a column is not positional, but rather is determined by its label. It follows from these

definitions that every item has a heading, a heading is a collection of columns, and an item has one value per column.

Observe that in this notion of an item there is no built-in distinction between the "real" data and the mere "attributes" of the data. E.g., we usually think of x,y,z coordinates of position as real data, and the time and entity they are associated with as attributes of the data; this conceptual distinction is not maintained in the representation of an item.

3.2 Notation

Here we introduce a convenient but equivalent alternative to the formal math notation above. Consider an item consisting of three column-value pairs, namely, a runIndex column of value 201 and integer type, a scenario column of value 667 and also integer, and a windDir column with value 0.25 and of type float. The expression for this item in the math notation used above is

$$\left\{\left(\left(\text{runIndex}, \text{Int}\right), 201\right), \left(\left(\text{scenario}, \text{Int}\right), 667\right), \left(\left(\text{windDir}, \text{Float}\right), 0.25\right)\right\}$$

To improve readability, we will use an alternative notation of label-value assignments, where the type can be inferred from the value (e.g., floats have decimal points, integers do not, arrays use nested square brackets, etc.). For example, the above item can be denoted as:

```
runIndex=201 scenario=667 windDir=0.25
```

Recall that since we are representing a set, the ordering of elements in an item is immaterial, and so the notation above is indistinguishable from:

```
scenario=667 windDir=0.25 runIndex=201
```

3.3 Logging Stream

An important property assumed of the logging stream is that each item is self-sufficient, independent of the context of other items. Consider this counterexample:

```
time=0.1
entity=e1
xpos=10.0
ypos=-5.0
zpos=90.0
entity=e2
  etc.
```

It would be very natural to read this stream where xpos, ypos and zpos pertain to entity e1, and that e1 and e2 data are both associated with time 0.1. In such an interpretation, the role of the time and entity items is not to be informative in themselves but to "change the subject" of what the stream is currently providing information about. As a result the order of items is critical; the meaning of an item depends not only on the item itself, but also on what items have come before it. Such a structure is perfectly reasonable for some applications but is not what we are defining here. An encoding of the same intention as a proper logging stream could rather look something like this:

```
entity=e1 time=0.1 xpos=10.0
entity=e1 time=0.1 ypos=-5.0
entity=e1 time=0.1 zpos=90.0
entity=e2 time=0.1 xpos=0.0
  etc.
```

Such a structure may give the impression of wasteful repetition, but we emphasize that this is an abstract data model, not a proposed implementation. Efficiency is important, but is a consideration downstream of the current analysis.

Item self-sufficiency is achieved simply by way of more informative headings. Significantly, the logging stream becomes order-independent; there is no notion of "what the stream is currently providing information about;" the stream is *stateless*. As a result, a subset of items can be selected and worked with on their own without the need to track down where they came from, and items from different sources can be interleaved into a single logging stream without concern for the stream's state. These are relevant considerations for *M*-on-*N* EW simulation and analysis.

3.3.1 Headings and Hierarchy

To carry out a hierarchical "factoring" from a collection of headings, analogous to that done above on the table in Figure 1, it is necessary that some columns from different headings have a common meaning – this is key to hierarchical structure.

For example, the "position" column might mean an xyz position in one model, but a Boolean indicating whether an entity is "in position" or not in another model – they mean different things, and the use of the same name is a mere coincidence. In this case the "position" column is not a candidate for factoring out. On the other hand, in the examples in this report, "entity" always means the same thing, and therefore it is a candidate for factoring. Questions to consider include, how do we indicate that different appearances of the same label have the same meaning? (one option is to simply *require* this), and, what does it mean if there is no column shared by all items?

3.4 Examples

In numerous cases below, alternative item representations encode exactly the same data but may be processed differently by a logger capability.

Here are several per-run data, where the designer chose to spread them out over several items:

```
runIndex=201
scenario=667
windDir=0.25
```

The same data in a design where they are condensed into one item:

```
runIndex=201 scenario=667 windDir=0.25
```

The choice between these has implications for what headings end up appearing in the logging stream.

A similar choice is made for per-entity data. Spread out over several items:

```
entity=e1 type=radar
entity=e1 model=notional
entity=e1 beamwidth=10.0
```

Or condensed into one item:

```
entity=e1 type=radar model=notional beamwidth=10.0
```

The bulk of items in the data stream will be stamped with an associated sim time:

```
entity=e1 time=0.1 xpos=10.0 ypos=-5.0  zpos=90.0
entity=e1 time=0.2 xpos=20.0 ypos=-10.0 zpos=90.0
entity=e1 time=0.3 xpos=30.0 ypos=-15.0 zpos=90.0
```

3.4.1 Heading Choices for Array Types

Commonly, some data should be grouped together or structured in a regular way. These arrays of data can appear in items in several ways. Next we will briefly consider several possible heading designs for array types and their implications. Among other things, these options can be interpreted as means to introduce inherently order-dependent array data into an order-independent logging stream.

Most simply, the value type may directly be an *array*. E.g., position data may be in a single "position" column whose type is 1-D array of length 3.

```
entity=e1 time=0.1 pos=[10.0, -5.0, 90.0]
```

The item may have one element per *component*. In this design, each datum has its own associated column. E.g., for 3-D position data, the heading could consist of three floating point numbers labeled x, y, and z.

```
entity=e1 time=0.1 x=10.0 y=-5.0 z=90.0
```

There can be one item per *coordinate*. The item heading includes columns for the component and the associated value. For the 3-D position example, a component column could take values like x, y, or z, and the data column would hold the associated numeric value.

```
entity=e1 time=0.1 component=x value=10.0
entity=e1 time=0.1 component=y value=-5.0
entity=e1 time=0.1 component=z value=90.0
```

It is common for the component to be an integer index.

```
entity=e1 time=0.1 index=0 value=10.0
entity=e1 time=0.1 index=1 value=-5.0
entity=e1 time=0.1 index=2 value=90.0
```

If the logging stream was not stateless, ordering alone could be enough to determine the indices of a sequence of items. However, to satisfy our definition of a logging stream, any ordering information should be encoded into index columns so that information does not change if the items are reordered.

For logical completeness we can consider an item-valued type (this corresponds most closely to a relation-valued attribute (RVA) in relational database theory [Date2005 p31]). Here we delimit the contained item in parentheses.

```
entity=e1 time=0.1 pos=(x=10.0 y=-5.0 z=90.0)
```

The direct array and components approaches are very similar. Both have a fixed number of elements, though the components representation does not have an explicit array shape and nothing necessarily constrains the components to all be the same type. In a relational table, an array can be converted to components with the extension and projection operations: first extend the array table with columns for the components and populate them with data sliced out of the array, then project away the array column. These steps can be reversed to revert from components to array.

The coordinates representation is quite different from array or components. Contrasting coordinates vs. components, when we wish to address the data as an array, i.e., via indexing, the indices (x, y, and z in the 3-D position example above) are in the columns of the components representation, but in the "rows" (item values) of the coordinates representation.

The array shape and number of elements is much more rigid in array and components representations than in coordinates. The coordinates representation thus can support ragged or sparse arrays "automatically" without additional fuss. A transformation from coordinates representation to array or components does not appear to have any brief expression in terms of elementary relational operations.

3.5 Query Notation

Here we sketch out some options for what a terse, expressive navigational or query notation of a logging stream might look like. (Considering the huge range of queries possible, the discussion is necessarily circumscribed.) The importance of a query notation is that it is the primary way the data gets accessed. (Data are also accessed by manual browsing of the files as a sanity check.) The notation's expressiveness and convenience are paramount.

Note that while the query notation may be closely bound to the data representation, this is not necessary in principle. The programming language in which the query notation is embedded is probably a more significant constraint than the underlying data model or representation. Design and implementation of an expressive query capability is both a worthwhile design goal, and a separate question from representation. The distinction is worth making for the additional degree of freedom it affords. However, if a representation (such as HDF5) comes with an acceptable query notation more or less built-in, so much the better.

3.5.1 Member-Index

A very succinct and familiar notation for navigating logged EW data, used to some extent in legacy solutions and very well-adapted to use in common scripting and programming languages, uses "dot" notation to select a member and square brackets to index an array. These notations, or constructs similar to them, appear in C++, Java, Python, and MATLAB. (Structurally this notation is very similar to "path" notation where a sequence of member names or indices is separated with forward-slashes.)

How does the logging stream construct relate to such a *member-index* query notation? It depends on recursive factoring of headings foreshadowed above in Figure 1, and which may be carried out as follows. At a given level in the hierarchy, identify an appropriate heading element and factor it out. This yields a collection of subsets of the logging stream, each selectable by a value for the factored column. Apply this operation recursively to each of the subsets until a "natural stopping point" is reached – perhaps when the remaining data has a highly structured array form. (Problems can arise if, e.g., a factorable heading does not exist.)

For example, we may factor a table on entity and then component, as was done in Figure 1. Example queries to this database could include `db.e1.xpos`, which signifies a query for the data associated with

entity e1 and component xpos, and would return an array with columns of time and val (with a value of xpos); and `db.e1.xpos[0]` for the 1x2 array containing the first time and xpos values.

3.5.2 Label-Value

The member-index approach becomes inconvenient if, for example, I want to select data for t=0.5. Syntactically, 0.5 is inappropriate for member access, and some languages will balk at its use as an index because it is not an integer. To address this we could introduce a *label-value* selector method, perhaps `lv(label,value)`. For example, a sample usage for selecting e1's xpos at time 0.5 might be denoted thus:

```
db.lv('entity','e1').lv('param','xpos').lv('time',0.5)
```

Besides supporting queries on floats like time, this notation also frees us from defining an ordering tree on the columns. It can do this because the columns are specified in the arguments, rather than being decided *a priori* to be in a certain order.

3.5.3 Discussion

In evaluating these makeshift query notations, we compare them briefly to the selection (a.k.a. restriction, filter, "discarding rows") and projection ("discarding columns") operations in relational algebra. Although the logging stream is not a table, the selection and projection operators have a natural, even obvious, generalization to it. The basic operation in both the member-index and label-value notations combines generalized selection and projection: only items matching a given value in a certain column are kept, then that column is projected away. Expressing a query in terms of selection and projection avoids some of the shortcomings of both notations above: it avoids the order-dependence of member-index, and unlike label-value it allows operations such as selecting columns from items designed in the component representation described earlier.

4. DATA MODEL ADAPTATIONS

In this section we consider how the data model of a logging stream defined above may be adapted to the relational and HDF5 data models.

4.1 Logging and the Relational Model

The relational model is well-known with an extensive literature and so will not be summarized here. Helpful background information may be found in [Mayne], [Welling] and many other sources. The definition of the relational model used here substantially follows [Date2005].

In the spirit of structural typing [Pierce, section 19.3] (or of duck-typing more loosely), observe that items with the same heading may be grouped together naturally to constitute a relation (in the relational database sense [Date2005 p45]) or table. Since we assumed above that each item in a logging stream is complete or self-sufficient, i.e., it does not depend on outside context or information such as ordering, the items of the stream can be "dealt out" to relations of the appropriate heading without a loss of information. So, for every heading appearing in a logging stream, let there be a corresponding table with the same heading. Then for each item in the logging stream, select a table according to a heading match, and add the item's data as a record in that table.

For example, consider this brief logging stream.

```
entity=e1 model=notional beamwidth=10.0
entity=e2 sensor=imager
entity=e1 time=0.0 mode=1
entity=e1 time=0.1 x=10.0 y=-5.0 z=90.0
entity=e2 time=0.1 x=0.0 y=100.0 z=10.0
entity=e2 time=0.1 image=[...]
entity=e1 time=0.125 mode=2
entity=e1 time=0.2 x=20.0 y=-10.0 z=90.0
entity=e2 time=0.2 x=0.0 y=200.0 z=10.0
entity=e2 time=0.2 image=[...]
entity=e1 time=0.3 x=30.0 y=-15.0 z=90.0
entity=e2 time=0.3 x=0.0 y=300.0 z=10.0
entity=e2 time=0.3 image=[...]
```

This stream has five unique headings, which corresponds to five tables:

```
entity, model, beamwidth
entity, sensor
entity, time, mode
entity, time, x, y, z
entity, time, image
```

After processing the logging stream the tables will have been be populated with item data. The tables with headings "entity, model, beamwidth" and "entity, sensor" will contain only one record each. The table headed "entity, time, mode" will have two records, that headed "entity, time, image" will have three records, and the table headed "entity, time, x, y, z" will have six records.

Note that value types in the logging stream are allowed to include composite data types such as tuples and arrays. We find persuasive a notion of the relational model in which such values are considered atomic [Date2005 p29] (as contrasted with a more traditional view, e.g., [Welling pg32]; to be fair, the traditional view is likely motivated more by existing DBMS implementations than by theoretical considerations).

A choice of headings in the logging data stream that leads to a useful collection of tables is a matter of database design. For example, should entity parameters be logged as a collection of small items, or one comprehensive item? Here this question is only addressed partially, in the abstract data model aspect.

At any rate, the collection of tables that results from this procedure is formally a relational database.

4.1.1 Views

While the tables described above are self-assembled according to a natural criterion, the resulting arrangement may not be the most convenient. In the example above, positions for both entities e1 and e2 are described with heading "entity, x, y, z," so the position logs of both entities were interleaved into one table. But the user may prefer one table per entity over this all-positions table.

However, this is not a fundamental limitation because per-entity position tables can be readily derived from the all-positions base relation. Such tables that are defined in terms of base tables are known as views [Date2005 p67]. For example, the position tables can be expressed with restriction operations on different entity values. Alternatively, they are the values in a grouping [Date2005 pg99], [Date2004 p203] of the base relation over entities. In this approach, a portion of the database design task is postponed from generation of the logging stream itself to a post-processing step.

4.2 Logging and HDF5

Next we will consider how the data model of a logging stream may be adapted to the HDF5 data model. This will be done after a brief introduction to HDF5.

4.2.1 A Brief Overview of HDF5

HDF5 is a collection of related things. Primarily this includes an abstract data model, a storage format, and an implementation. In addition, it also includes a collection of command-line tools, and extensions for other languages.

The abstract data model defines how we conceive of the data's organization, and guides how we work with the API. The storage format is something we never expect to bother with; the point of using a standard format is to avoid writing our own generators and parsers, relying on a library to do that for us. (However, developing confidence in this storage format is an important milestone in building confidence and acceptance of the format.) The implementation is available for numerous platforms, and presents a C API. Extensions for other languages such as C++, MATLAB and Python greatly increase the usefulness of the format: different tools written in different languages can collaborate on a common problem. At a beginning level, command-line tools are important for sanity checks, for a first-look at data received from collaboration partners, for evaluating legacy data, and for gaining insight into the HDF5 data model by looking at examples.

4.2.2 HDF5 Abstract Data Model

In its data model, an HDF5 file consists of groups, datasets, and attributes. Groups contain datasets or other groups; they are what puts the "H" (hierarchical) in HDF (hierarchical data format). They are akin to a file system folder or directory. A dataset is a homogeneous *n*-dimensional array of some type. Finally, both groups and datasets may have user-defined attributes, which we think of as a collection of key-value pairs.

Groups and datasets have names. The "path" to a group or dataset uses the same forward-slash notation as a Linux file system. Every HDF5 file contains a single root group.

As observed in [Rossant] (in which the author describes why his group abandoned HDF5),

> "A simpler and roughly equivalent alternative to HDF5 would be to store each array in its own file, within a sensible file hierarchy, and with the metadata stored in JSON or YAML files."

This provides a concrete and illuminating analogy for understanding the HDF5 data model.

4.2.3 Logging and the HDF5 Model

Now we will analyze the logging stream to see how it may be conformed to the HDF5 model. In this analysis we will focus on groups and datasets, and set aside attributes for simplicity.

We need to determine what datasets there are, where they are located in the group hierarchy, and what their names are. Then we will consider the dataset arrays as objects to be "grown" as the logging stream is processed. This is analogous to the "building up" or accumulation of items to tables in the relational case.

The HDF5 distinction between groups and datasets induces a distinction in item elements. Some of the elements contain data that will end up stored in a dataset. The only use for the elements that remain is to determine which dataset the associated data belongs in; we will call these elements table-selectors.

The mapping from table selectors to a dataset in a group hierarchy may be arbitrary, and is an HDF5 layout design question, akin to a database design. This mapping is the primary adapter from the logging stream data model to the HDF5 data model. One plausible means to lay out this mapping in a way that takes its cue from the data is to follow a recursive factoring procedure akin to that described above for the notional member-index notation. From the headings that appear in a logging stream, factor out an appropriate table-selector heading element. For each value taken by this element, instantiate an HDF5 group at this level. Continue recursively until the table-selector heading elements are exhausted; what remains is array data. At each such "leaf node" instantiate an HDF5 dataset to contain the array data, where name and dimension are decided by some appropriate means.

For example, using the same logging stream as the relational model above, the only clear table-selector is `entity`. After this, we have candidates for dataset representation: position and mode for e1, and position and image for e2. The resulting paths of the datasets could be:

```
/e1/mode
/e1/mode_time
/e1/position
/e2/position
/e2/image
/e2/image_time
```

We have chosen to include time as a column in the position datasets because the data are all floats, but to separate it out to another dataset for mode and image, because they are different types. The tables with one record only are good candidates for HDF5 attributes. This example suggests that a simple general rule for mapping from items of a logging stream to HDF5 structure is not straightforward.

While the query and representation models are independent in principle as noted above, there is an obvious advantage if the resulting layout can be adequately navigated using built-in HDF5 query capabilities only: no separate (and potentially inefficient) query algorithms need to be developed and maintained.

5. HDF5 EXPERIMENTAL RESULTS

To complement the theoretical analysis documented above, several small-scale implementation experiments were carried out. For a work-in-progress report such as this it is sufficient to summarize the results.

Demonstrated capabilities include: adequate handling of a large structure, i.e., an array that used more than half the memory available on the system; building up an array in portions; interleaved writing to separate arrays in the same file, and later access to each of the arrays as a whole; adding a README attribute to an existing file; and translation of sample code from C to Python using the low-level API in the latter.

Additional small-scale experiments planned include complex numbers, a sequence of arrays of different sizes, and creation of new groups and datasets after some data has been written.

6. CONCLUSIONS

The two main conclusions that can be drawn at this time are:

- At the abstract data model level, both a relational database and HDF5 can be used for logging the output of EW simulations
- Regarding HDF5 more broadly, the investigation did not turn up any obvious disqualifications. It is a promising technology that deserves further investigation into how it may best serve the needs of logging of EW simulations.

Additionally, this work emphasized the importance of distinguishing data representation from query notation.

In spite of its plausibility at the abstract data model level, we recommend against the adoption of relational DBMS software for EW simulation logging at this time for the following technical reasons.

1. The adaptability of a logging stream to the relational model argued above assumes a relatively evolved concept of the relational model which many DBMS implementations do not adhere to.
2. The current user base is very experienced with array manipulation and comfortable with hierarchic navigation (at a minimum on the analogy to navigating a file system), but has little to no experience with formulating queries in the SQL query language usually used in relational DBMS.
3. The data itself is reasonably well-adapted to the HDF5 model, and many end-user operations are array operations.
4. Much of the benefit of a DBMS lies in relational join operations, query optimization, integrity of updates (transactions), rollback, and related features, which are important to mainstream DBMS applications but largely irrelevant to write-once, read-many scientific data, especially the output of computer simulations.

Some HDF5 features of potential interest were consciously ignored in this investigation, including compound data, hard links, soft links, external links, usage from MATLAB and C++, dimension scales, the HDFView GUI tool, and parallel I/O.

Some issues related to the concerns of this report arise when sharing data with other users or organizations. If we are the data's only user, we are fairly free to construct an *ad hoc* structure on the fly, but collaboration requires conventions. A custom file format requires custom parsers and writers, which creates a deployment and maintenance burden. With HDF5 this particular responsibility is relieved, but because of its flexible nature we are still left instead with the higher-level, more limited design task of agreeing on a convention for groups, datasets, and attributes. Also, as noted in [Collette], "Because HDF5 is self-describing, anyone using Python, IDL, MatLab or any of a dozen other environments could simply open up the file and poke around." Such a consideration is significant in an experimental research environment.

Additional, miscellaneous conclusions and observations include the following.

- Logging is only one possible role for HDF5 in EW simulation. It is common for simulation models to depend on large static data sets. This data may be encoded in custom file formats and the files may be so large that they cannot be loaded into memory in their entirety, but must have sections paged in and out. Such cases are candidates for delegation to HDF5.
- Treating data as functions – interpolation is a very direct example – tends to be underrepresented and deserves more consideration in scientific database discussions. The extent to which this viewpoint may be addressed in HDF5 has not been fully explored.
- It was briefly observed that some relational operations have natural, even obvious, generalizations from a relational table to the logging stream. This may be a fruitful avenue for understanding better the formal similarities and differences of the logging stream and relational models.
- A hierarchical navigation query style appears very natural for our application. Yet the hierarchical model as a whole was tried and abandoned in mainstream practice in favor of the relational. This poses a nagging question of whether our domain of scientific computing is truly different enough to indicate a different database solution, or whether we are slow to see how to address our problems in terms of a superior technology.
- This report largely restricted itself to formal aspects of the logging problem, and many substantive issues remain to be explored.

7. ACKNOWLEDGEMENTS

We are grateful to Eric Fischer and Brook Susman for helpful technical discussions.

8. REFERENCES

[Folk] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, Dana Robinson. "An Overview of the HDF5 Technology Suite and its Applications." AD '11 (Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases), pages 36-47, Uppsala, Sweden, March 25, 2011.
https://doi.org/10.1145/1966895.1966900

[Date2005] C.J. Date. Database in depth : relational theory for practitioners. O'Reilly, 2005.

[Welling] Luke Welling and Laura Thomson. MySQL Tutorial. Sams Publishing, 2004.

[Silberschatz] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. Database System Concepts, Sixth Edition. McGraw-Hill, 2010.

[Mayne] Alan Mayne, Michael B. Wood. Introducing relational database. Manchester : NCC Publications, 1983.

[Date2004] C.J. Date. An introduction to database systems, 8th Edition. Pearson/Addison Wesley, 2004.

[Pierce] Benjamin C. Pierce. Types and programming languages. Cambridge, Mass. : MIT Press, 2002.

[Rossant] Cyrille Rossant. “Moving away from HDF5.” January 6, 2016. http://cyrille.rossant.net/moving-away-hdf5/ Visited 2/8/2017.

[Collette] Andrew Collette. “HDF5 as a zero-configuration, ad-hoc scientific database for Python.” March 25, 2015. https://hdfgroup.org/wp/2015/03/hdf5-as-a-zero-configuration-ad-hoc-scientific-database-for-python/ Visited 2/8/2017.